Memory Allocation

Memory Allocation — dynamic

= allocate + free memory chunks of arbitrary size at arbitrary points in time

- almost every program uses it (heap)
- don't have to statically specify complex data structures
- can have data grow as function of input size
- kernel itself uses dynamic memory allocation for its data structures

implementation: has huge impact on performance, both in user and kernel space

fact: it is impossible to construct memory allocator that always performs well
$\rightarrow$ need to understand trade-offs to pick good allocation strategy

Dynamic Memory Allocation — principle

initial: pool of free memory

tasks:

- satisfy arbitrary `allocate` + `free` requests from pool
- track which parts are in use/are free

restrictions:

- cannot control order/number of requests
- cannot move allocated regions $\rightarrow$ fragmentation = core problem!

Dynamic Memory Allocation — bitmap

idea:

- divide memory in allocation units of fixed size
- use bitmap to keep track if allocated (1) or free (0)

problem: needs additional data structure to store allocation length (otherwise cannot infer whether two adjacent allocations belong together or not from bitmap)

Dynamic Memory Allocation — list

method 1: use one list-node for each allocated data

- *extra space* needed for list
- allocation lengths already stored

method 2: use one list-node for each unallocated data

- can keep list in unallocated area (store size of free area + pointer to next free area in free area)
- *additional data structure* needed to store allocation lengths
- can search for free space with low overhead

method 3: both

Dynamic Memory Allocation — problems

fragmentation is hard to handle

factors needed for fragmentation to occur:

- *different lifetimes*
- *different sizes*
- *inability to relocate previous allocations*

all fragmentation factors present in dynamic memory allocators!

Allocation — best fit vs. worst fit

idea: keep large free memory chunks together for larger allocation requests that may arrive later

best-fit: allocate smallest free block large enough to store allocation request

- must search entire list
- *problem*: sawdust — remainder so small that over time left with unusable sawdust everywhere
- *idea*: minimize sawdust by turning strategy around

worst-fit: allocate largest free block

- must search entire list
- *reality*: worse fragmentation than best-fit

Allocation — first fit

idea: if fragmentation occurs with best and worst fit, optimize for allocation speed

principle: allocate first hole big enough

- fastest allocation policy
- produced leftover holes of variable size
- *reality*: almost as good as best-fit

First Fit — variants

first-fit sorted by address order

LIFO first-fit

next fit

Allocation — buddy allocator

idea: allocate memory in powers of 2

- all chunks have fixed 2^n-size $\rightarrow$ allocation request rounded up to next-higher power of 2
- all chunks naturally aligned

no sufficiently small block available:

- select larger available chunk, split into two same-sized buddies
- continue until appropriately sized chunk is available

two buddies both free (2^n): merge to 2^{n+1}-chunk